FR-8047

Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems

D.L. Parnas

June 3, 1977

Naval Research Laboratory

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 8047	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* USE OF ABSTRACT INTERFACES IN THE DEVELOPMENT OF SOFTWARE FOR EMBEDDED COMPUTER SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Interim report on a continuing NRL Problem 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) DAVID L. PARNAS		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, D.C. 20375 Code 5403		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NRL Problem B02-18 Project -XF21-241
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Research Laboratory Washington, D.C. 20375		12. REPORT DATE June 3, 1977 13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Naval Electronics Systems Command Arlington, Virginia 20360		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Software	Abstract interfaces
Software engineering	Interfaces
Abstraction	Programming
Embedded computer systems	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes a procedure for designing computer systems that are developed specifically to be a component of a more complex system. Two significant characteristics of such design problems are the following: the computer system interface is determined by factors outside the control of the computer system designer, and the specifications of that interface are likely to change throughout the life cycle of the system. The purpose of the procedure described in this report is to reduce "maintenance" costs by means of a software organization that insulates most of the programs from changes in the interface. The procedure is based on the systematic compilation of an *assumption list*. The assumption

(Continued)

20. Abstract (Continued)

list describes those aspects of the interface that future users and other knowledgeable persons consider essential and therefore stable. Other aspects of the interface are ignored. An *abstract interface* is designed on the basis of this assumption list. A specification of the abstract interface is used to procure the major components of the system.

This report explains the principles behind the procedure and illustrates its use. The success of the procedure is primarily limited by the ability of designers and future users to compile an accurate list of assumptions. A side benefit of the procedure is simpler, better structured software. Successful application of the procedure should result in both increased reliability and reduced lift-cycle costs.

CONTENTS

USE OF ABSTRACT INTERFACES IN THE DEVELOPMENT OF SOFTWARE FOR EMBEDDED COMPUTER SYSTEMS

INTRODUCTION

This report describes an approach to software design and procurement that should be useful in the acquisition of software for embedded computer systems. We will refer to a computer system as *embedded* whenever it is specifically developed to function as a component of a significantly larger system. This is intended to distinguish embedded computer systems from computer systems that are developed as general mechanisms to be used in vaguely specified applications. A "general purpose" operating system is an example of a nonembedded system; the "message processors" developed for use in communication networks are good examples of embedded systems.

Although we cannot precisely delineate embedded and nonembedded systems, the systems with which we are concerned have the following characteristics:

- The designer of the embedded computer system is not free to define the interface to his system. He is required to meet an interface that was determined by factors beyond his control. For example, he cannot define the input language or specify the character set to be used.

- The constraints placed on the computer system by the interface requirements are strict and often quite arbitrary. The external system is not tolerant of deviations; a system may come very close to meeting the requirements and still require very extensive modifications before it can be used.

- The interface often changes during the period in which the computer system is developed. The system in which the computer system is embedded may be being developed at the same time, or it may be undergoing evolutionary changes while in use. Because the computer system is only one of many components, the effects of any changes on the computer system are given relatively little consideration.

- Often there are several similar systems with similar requirements, but the interface requirements are so strict that it is not practical to modify one computer system to replace the other. This may happen because of evolutionary changes or because two different contractors have "total system" responsibility. In such cases, there is a great deal of duplicated effort.

This description of an embedded system is somewhat broader than the standard military definition of "embedded." We include other systems which have many of the same problems. Many of the statements we make hold for a broader class of systems, but the problems are more acute in embedded systems.

Manuscript submitted April 13, 1977.

These characteristics place those who must write software specifications in a dilemma described by the following statements:

- Those who will produce the software must be provided with a precise description of the requirements that the software must meet. Without a precise specification, the chances that the product will be statisfactory are low; without specifications, one must depend on the good will of the software developer, because one cannot prove that a product is defective.

- The details of the interface must be considered unknown. It is almost certain that the requirements that one could describe in a contract will not be the same as the requirements that must be satisfied when the system is used.

- Systems developed to meet old interfaces are often surprisingly hard to adapt to the current interface. Many early design decisions have been based on information which is no longer valid [1]. Finding those portions of the code that must be changed is difficult, time consuming, and expensive.

- Finding an alternative source of supply for changes is unlikely. Knowledge of implementation details is needed to make changes. Competition does not hold the price down.

The subject of this report is one way to escape from this dilemma. This report proposes an organization of the software that results in divorcing the majority of the code from the tight constraints.

EXAMPLES

To illustrate the problem, we describe three examples of embedded computer systems, emphasizing the reasons that the interfaces can be expected to change.

A Message-Forwarding Station for an Existing Communication Network

A not unusual application of a computer is to automate the work of the human operator at a relay point in a communications system. The operator must observe incoming messages and detect those that require action on his part. In addition to delivering messages to addressees, he must keep logs and assist in the preparation and transmission of outgoing messages. Many communication networks have complex coventions for identifying and routing messages. The conventions are especially complex if the channels in use may be noisy. Often the conventions have evolved to a point where they appear completely arbitrary and capricious. Nevertheless the conventions must be strictly observed or messages may go astray. If a human operator at one station is replaced by a computer, the system conventions will not change. The computer will have to meet the same interface as the man did. Most improvements in routing conventions, etc., will be made to increase communication effectiveness (priority schemes, etc.), not to make programming easier. Although the functions to be performed will not change much, the interface can be expected to change repeatedly both during the programming and after the system becomes operational.

Radar Data Analysis

Computers are often used to process data obtained from radar units to prepare displays for human operators and detect significant events. The requirements to be met by the computer system are determined primarily by

- physical laws (propagation characteristics, gravity, air resistance),
- radar and associated communications technology,
- traffic conventions (such as minimum safe distances),
- human characteristics (these are man-machine systems), and
- display technology.

It is significant that "computer characteristics" is not in this list of determining factors; the computer system is expected to adapt to an interface constrained by these factors, not vice versa. Although the physical laws can be assumed to stay constant, the other factors can and do change.* Improvements in radar technology, traffic patterns, etc. will not be renounced in order to save the cost of computer system revision.

Address-List Processors

A somewhat less obvious example of an embedded system is a system to process address lists stored on tapes or other files. Postal-system conventions determine the addressing conventions. The interface conventions are not quite as strict or arbitrary as in the above examples, but they must nonetheless be observed. One can easily imagine the reaction of a U.S. postman to a letter addressed according to the German convention: city before street, zip code before city, house number after street. Our postal system would attempt to interpret the house number as a zip code, the street name as a town, the town name as a street name, and the zip code as a house number. Moreover the conventions change. A change in the German system (to place street before town) was recently announced. Since address lists are often purchased from a variety of sources, there are many input formats.

APPLYING THE *INFORMATION HIDING* PRINCIPLE WHEN EXTERNAL INTERFACES MAY CHANGE

References 1 through 3 have introduced a guideline for use in making the early design decisions in software design — particularly those decisions that determine the decomposition of the system into components for independent design (and later independent modification). This has been called the *information hiding* principle. Essentials of the procedure suggested are the following:

1. Identify a list of design decisions for which change cannot be ruled out (data structure, algorithms, etc.).

*Some readers may object to the statement that "human characteristics" will change. Although human beings as a class may stay the same, it is not infrequent to replace one class of operators with another (for example, to replace college educated engineers with specially trained technicians). Further, our understanding of the best way to communicate with humans may improve, and this is effectively the same as a change in the characteristics of the operator.

2. Make each design decision the "secret" of one module. In other words, the programs that cannot be coded without knowledge of this decision comprise a module. No program is in two such modules.

3. Design the module interface. The interface consists of the "subprograms" needed by the module user in order to make use of the module's data structures and algorithms without knowing the design decision that is being hidden. This interface is so designed that it can be kept unchanged even if the data structures or algorithm must be revised. The set of programs is kept minimal in the sense that only those that cannot be efficiently performed without direct access to internal data are in the module. Most of the tasks that require the data are performed using the interface functions.

Readers who are encountering this idea for the first time should read Refs. 2 through 4 before continuing.

The information hiding principle was developed and presented as a means of reducing the cost of changes in *internal* design. Here external aspects are likely to change. The information hiding principle suggests a system structure in which those aspects of the external interface that are likely to change are hidden from the bulk of the system.

Such a system would have a structure such as that shown in Fig. 1. The large box represents those programs whose function is not dependent on volatile details of the interface. The small box has the changeable aspect of the world as its "secret." The designer can now design the interface to the large box relatively free of the constraints associated with embedded systems. Diagrams such as Fig. 1 are much easier to draw than they are to realize. We have not yet demonstrated that such an organization is feasible; we have not shown how to design the system. We have only reformulated the problem.

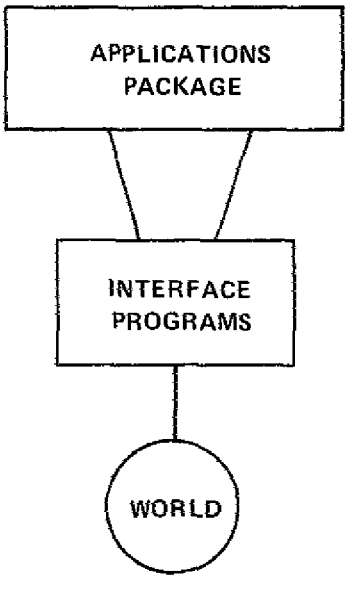

Fig. 1 — Structure of a system formulated according to the information hiding principle

Our new formulation allows us to propose the following procedure:

1. Specify the "internal" interface, that is, the interface between the little box and the large box.

2. Obtain an implementation of the large box. The software producer considers the large box to be the whole system.

3. Near completion of the large box (or whenever the actual interface is really known) specify the small box. The software in the small box is constrained to use only the specified interface to the large box. It is not allowed to modify code or refer to data structures that are part of the large box.

4. Future changes in the external interface should result in changes to the small box but not the large box.

Again we have reformulated the problem but not solved it. A skeptic may well ask how we can make 4 come true. We have not shown why such an artifically derived interface is preferable to our best guess about the actual interface. That is the subject of the next three main sections.

"ABSTRACT" INTERFACES

What is an Abstraction?

In recent years the word "abstract" has become one of the buzzwords of the computer field. In some discussions it is used euphemistically to mean vague, unrealistic, or insufficiently specific. In others it is used to mean formal, highly mathematical, etc. In this report we refer to something as an *abstraction* if it represents several actual objects but is disassociated from any specific object.

It is the many-to-one relationship that is critical. For example, differential equations are one mathematical abstraction that can represent many real systems as diverse as RLC circuits and collections of springs and weights. An abstraction represents some aspects of the system but not all. Consider a map as an abstraction representing a road network. This graph may represent the lengths of the roads, but not the type of pavement or colors. Thus, one such graph could represent many different road systems, including both black or green, asphalt or concrete. The common aspects of the road systems (the lengths of the various road segments) are represented; their differences are not.

Why are Abstractions Useful?

If all properties of the abstract system correspond to properties of the real system, then we can learn about the real system by studying the abstraction. Everything that is true about the abstraction corresponds to some fact about the real system, although the reverse need not be true. The abstraction is usually easier to study. It is far easier to find a good route by studying a road map than by exploring the road network itself. Abstractions are simpler and more simply described than the actual objects. With the proper abstractions, we ignore all of

the details that are not relevant to our analysis. More important, any result that we obtain by studying the abstraction can be reused! It can be applied to other systems that the abstraction represents. For example, mathematical results obtained by studying the equations representing electrical circuits were later applied to the study of electrical motor systems. Directed graphs provide another example. An incredible variety of problems have been solved by representing the system as a graph and applying well-known algorithms to find the shortest path, prime cycles, etc.

What is an Interface?

It is often assumed that the interface between two software components may be described by describing the format of the information that they exchange. This is a gross oversimplification which has resulted in a great many expensive errors. A complete description of the interface must include a statement of *all* of the assumptions that each component makes about the other [3]. Anything less is not a complete description of the ways (intended and unintended) that the two components might interact. The list of assumptions usually includes an explicit description of the *intended* interactions; unintentional interactions can occur if one of the components violates an assumption that the other makes.

A description of the formats used for information exchange does not describe all of the assumptions. Assumptions about the meaning of the information, resource usage, etc. must also be described. In fact, one can describe an interface without describing the formats of the information exchanged. One can define a set of programs to be used for inserting and accessing information. One then describes the way that these programs influence each other's behavior. This can be done without describing the data structure that is used [4]. The definition of these procedures or programs is a part of the description of the interface between any components that use the programs to communicate. A description of the formats used by those functions need not be included, because neither component makes any assumptions about the format. We repeat: an interface description is a decription of a set of assumptions. The description of an interface between several programs is not complete unless all of the assumptions that the programs make about each other are included.

What is an Abstract Interface?

We use the phrase *abstract interface* to refer to a set of assumptions that represents more than one possible interface. An abstract interface will model some properties of those interfaces that it represents but not all. It will describe their common aspects while hiding (or ignoring) the differences. It will not generally be sufficient to permit development of a working system.

As with any valid abstraction, all facts that are true of the abstract interface are true of any one of the actual interfaces that it represents. It follows that any program that can be demonstrated to be correct using only the information about the abstract interface will be correct (usable) for *any* of the real interfaces represented by the abstract interface.

However the information in the abstract interface, being restricted to that information that is true for many distinct actual interfaces, is not generally sufficient for the writing of a complete program. Any programs that we write and verify, using only that information implied

by the description of the abstract interface, will not be incorrect. They will however assume the availability of programs that cannot be written without additional information. These programs are being written by "stepwise refinement" [5,6]. They can be completed by adding programs that use the additional information specific to the actual interface and not true for *all* interfaces represented by the abstract interface. Those programs constitute the small box in Fig. 1.

In summary, the procedure that we are discussing can be formulated in yet another way:

1. Specify an abstract interface embodying all the information shared by all of the possible actual interfaces;

2. Procure programs to meet this abstract interface (the large box of Fig. 1);

3. Procure additional programs in order to meet the actual interface (the small box).

A change in the actual interface that does not violate assumptions made in step 1 can be made without changing the programs in 2. Step 3 must be repeated whenever such changes occur.

A SIMPLE EXAMPLE: A SYSTEM TO PROCESS DATES

The procedure being discussed can be illustrated by considering the problem of writing a program that will read in a date from some input media, compute, and print out the date 3 weeks from the input date. There is no standard format for representing dates. Among the many ways of representing dates are:

February 10, 1941	(month day in month, year),
10 February 1941	(day in month month year),
10 February 41	(day in month month last two digits of year),
10.2.1941	(day in month.integer encoded month.year),
2/10/1941	(integer encoded month/day in month/year),
41.2.10	(last two digits of year.integer encoded month.day in month),
41 February 10	(last two digits of year month day in month),
41,41	(day in year, last two digits of year).

Not only are there many formats, but it is impossible to look at a date and be certain which format it is in. Consider 10.11.12 or 12 November 10.

In spite of the variety of possible input formats, the algorithm for calculating the new date need not change if the format changes. It must be possible to organize the program as suggested by Fig. 1.

Compiling and Checking the List of Assumptions

The first step toward defining the abstract interface (the interface between the two boxes in Fig. 1) is to list assumptions that we may safely make about *all possible* input formats:

- It will be possible to calculate the year of the input date. If the two-digit encoding of the year is used, there will be no doubt about which century is intended. (If anyone is foolish enough to violate this assumption, the software designer cannot help him.)

- It will be possible to calculate the month of the input date.

- It will be possible to calculate the day of month of the input date.

We have used the phrase "it will be possible to calculate" rather than "the input will contain" so that our assumptions will be valid even if one changes to the use of a Julian date (41, 1941), or to some cryptic encoding of the date.

The "list of assumptions" is intended to be a complete list of all that we need to know about the interface* in order to write the bulk of the code for the system. Such a list should be checked to make sure that it has neither excess information nor insufficient information. Excess information is information that is either (a) not needed to design the system or (b) not known at this time. There is insufficient information if some major portion of the system cannot be implemented without making additional assumptions.

It is possible that a list of assumptions has insufficient information because, were the information provided, it would be type-b excess information. Under these circumstances the information must be designated as a parameter of the design. The design should be carried as far as possible in terms of this parameter and without assuming a specific value for it.

If we view such an assumption list in this light, innocent looking statements will be found to have rather far-reaching implications. Consider the following example:

"*The message will contain a string that is a unique identifier of the message.*"

The implications of including this statement in an assumption list are the following:

- In all possible formats it will be possible to find a string that is the unique identifier mentioned.

- The bulk of the system's code can be written without knowing *how* to find that string in the data.

- The bulk of the system's code can be written without knowing any more information about the string (its length, that it is an integer, that H never appears, or whatever). *If we give no information about an item except that it is a string, we are stating that the item's structure is unimportant.*

- The system will never need to distinguish between two messages with the same identifier. All code may be written assuming that the identifiers never repeat (that a message is never sent twice).

- Without the assumption of the existence of such a unique identifier, the bulk of the system's code could not be written.

It is important that the reviewers of such statements recognize that the assumption list will be used to draw up a specification that will be given to developers *instead of* information

*The characteristics of the calendar (such as, 30 days hath September...) are not considered part of this interface. If such assumptions are likely to change, we will hide them in a separate module.

about the actual interface. Insufficient information in the assumption list will mean that important functions will not be implemented. Excessive information means that the system might be unnecessarily restricted in its applicability. Highly critical, careful review is essential.

Designing the Interface

The four assumptions about dates above allow us to assume the ability to implement integer-valued input procedures, YEAR, MONTH, and DAY that return the year, month of year, and day of month, respectively. A description of these integer procedures *and the assumptions about their meaning* constitutes the abstract interface. All programs in the large box of Fig. 1 can be written in terms of those procedures. If a new actual interface is encountered, a new implementation of those procedures will be needed. However, as long as our assumptions about the input remain valid, the remainder of the program need not be changed.

SUMMARY OF THE PROPOSED METHODOLOGY

The date-processing example given in the preceding section, though extremely simple, illustrates the main steps of the methodology that we are describing. The key to this method is a departure from the standard view of an interface as a set of formats for data communication. Instead we consider an interface to be defined by the set of assumptions that the components make about each other. Recognizing that an abstraction is something that represents many instances, we base the design of an abstract interface on that subset of the assumptions represented in the various actual interfaces that is true for all actual interfaces. Usually, the various possible interfaces have too little in common to allow complete programs to be written on the basis of these assumptions. The assumptions are sufficient to allow us to describe the syntax and semantics of a set of functions that can be implemented using additional information about an actual interface. Application programs written in terms of these functions are valid and usable for all input formats that satisfy the stated assumptions. The programs written to implement these input functions are specific to the particular interface and must be changed whenever the actual input format changes.

When an embedded computer system is procured and the actual interface to the computer system is not known, the major purchase should be a system that meets only the abstract interface. The contractor should be given precise specifications for the abstract interface and should be required to build a system that will work using any valid implementations of the functions supplied to him. In a typical system most of the code will be in that portion of the system that assumes the availability of those functions; the programs needed to implement the interface functions are small. Procurement of these additional programs can usually be delayed until the main part of the system is almost ready to be used. At this point, one usually knows enough about the actual interface to write a complete specification. Since precise specifications of both the actual and the abstract interfaces are available, the coding does not require knowledge of the internal structure of the remainder of the system. Competitive procurement may be used.

The success of this method depends on

- our ability to anticipate changes or variations sufficiently well that the assumptions made in defining the abstract interface prove valid for the actual interface (the oracle assumption) and

- the various possible interfaces having enough in common so that the additional programs needed to meet the actual interface are significantly smaller than the remainder of the system (the big-large-box assumption).

If these conditions do not hold, the methodology described may be of little help.

The use of abstract interfaces does not give the implementor of the large box more freedom than a conventional approach; it constrains him more tightly. He is prevented from making assumptions about the actual format. Even if the same contractor eventually makes both parts, we are forcing him to make his system better structured by defining this "internal" interface.

A LARGER EXAMPLE: A SYSTEM TO PROCESS MAILING LISTS

This section demonstrates the application of abstract interfaces on a more realistic example. The specifications and design were developed by John Guttag, Barbara Trombka, John Shore, David Weiss, and the author.

Many organizations maintain lists of addresses. In simple applications the whole list is used to generate a set of mailing labels or "personalized" letters. Other applications involve the selection of addresses of people who are more likely to be interested in the contents of the mailing. For example, an advertiser who wished to offer a new magazine called Tax Loopholes, might want to select those addresses that indicate a medical degree. Other advertisers might want to select all entries within a certain geographic area or to select those addresses with specific first or last names.

The lists themselves are obtained from many sources and are generally delivered on a medium such as magnetic tape in a format that corresponds closely to the actual printing format of the label. Lists obtained from different sources will not usually be in the same format.

A mailing-list processing system is an example of an embedded system (one subject to arbitrarily changing constraints), albeit one in which the constraints on the system are not as strict as in some other situations. The input format is determined by the systems that produced the tapes; the output format is considerably constrained by the requirements of the postal systems in which the mail will be deposited.

In this example we assume that we are dealing with mailing lists in which all of the addresses are of persons within government organizations. Even with this assumption, we cannot assume the input or output formats to be known. Figures 2 and 3 show two possible formats for addresses — the formats are defined by "fill in the blanks" forms. Were we to procure a large set of programs to process address lists, we would want to reduce the likelihood of major changes in the programs being caused by predictable* changes in the input or output format.

To apply our abstract interface methodology, we must list those properties of the addresses to be processed that can be expected to remain true. This list of assumptions would then be circulated to all concerned for approval or rebuttal. For our example an intial list of assumptions might be as shown in Fig. 4.

*In most situations a *predictable* change is one that the designer had no good reason to consider impossible.

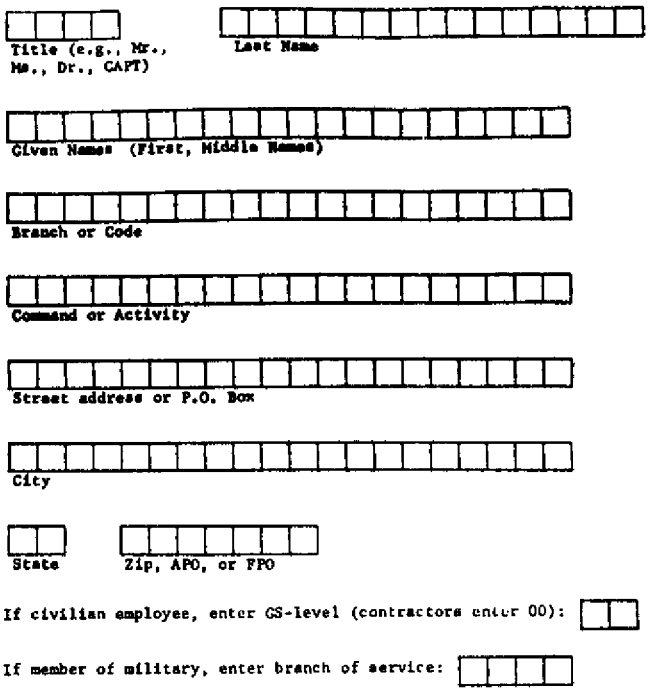

Fig. 2 — Possible format for addresses

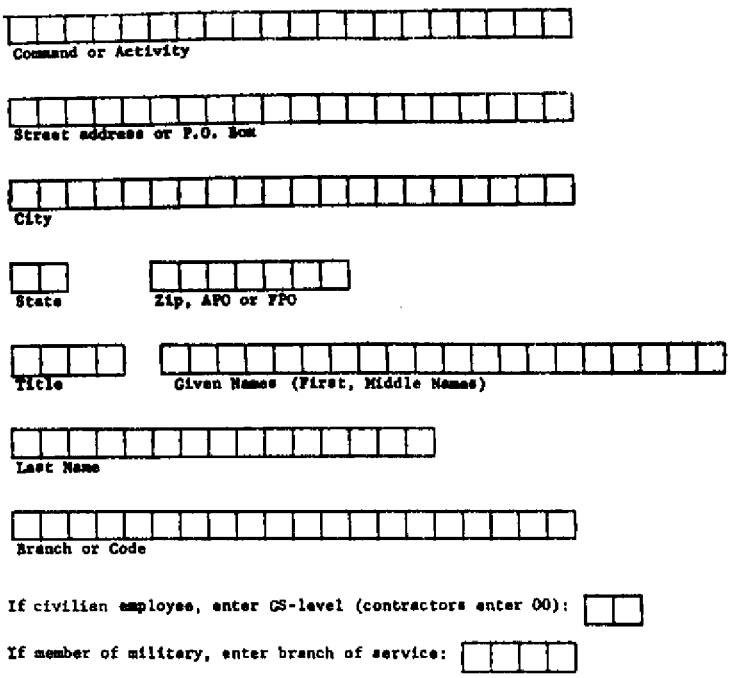

Fig. 3 — Possible format for addresses

The following items of information will be contained in addresses and can be identified by analysis of the input data; this information is the only information that will be relevant for our computer programs:

- Last name
- First name
- Organization
- Street address
- City, state and zip code (single line with a comma between city and state)

Fig. 4 – Typical initial list of properties that are assumed to remain true

We would hope that after such a list of assumptions was circulated, the following objections would be noted:

- Middle names may also be relevant (for example, to distinguish John Lyle Smith from John David Smith).

- An identifier of an internal mail post or other internal organizational division (such as, Code 5403, Information Systems Branch, etc.) may be relevant to some of the programs.

- A title such as Capt, Prof, Dr, Maj, etc. may be included in the address and be relevant for some purposes. It is important that name and title not be confused.

- Not all addresses contain the city and state and zip code in the single-line format as described. Certain input data may contain the zip code at some other point in the address. Further, in the addresses of military units overseas (APO NEW YORK 09175), the format is not valid at all. Even if one is willing to construe APO as a city in New York, the comma is not present.

- Certain organizations demand that civilian employees include their civil service grade, and this information may also be relevant for some of the processing.

- Some data sets may contain the branch of service for members of the military. This information is necessary in some applications because the title has different significance in different services. (Compare the treatment given Marine captains with that given Navy captains.)

- Many of the above items will often be absent from an address.

These errors in Fig. 4 are simply indications that the addresses examined in drawing up the list of assumptions had some properties that were not common to all the addresses that might be encountered. The exceptions might even have been known to the person who drew up the list, but he overlooked them. One purpose of compiling a list of assumptions and obtaining the approval of others in the organization is to increase our chance of finding such oversights at an early stage, before they can do much harm.

A more realistic list of assumptions is shown in Fig. 5. These assumptions identify the information that may be found in an address but make absolutely no statement about the position within the address at which it may be found. A program that could be demonstrated to be

The following items of information will be found in the addresses to be processed and constitute the only items of relevance to the application programs:

- Last name
- Given names (first name and possible middle names)
- Organization (Command or Activity)
- Internal identifier (Branch or Code)
- Street address or P.O. box
- City or mail unit identifier
- State
- Zip code
- Title
- Branch of service
- GS grade

Each of the above will be strings of characters in the standard ANSI alphabet, and each of the above may be empty or blank.

Fig. 5 — List of assumptions that is more realistic than the list in Fig. 4

correct making only the assumptions given in Fig. 5 could be used on an address file that contained addresses in scrambled formats. The assumptions are that the information can be found and that no other information will be needed; they do not tell us how to find the information or put any constraints on possible values.

The assumptions in Fig. 5 tell us that, if we have a file of addresses sequentially numbered, we can implement functions such as (a) FETTIT(i), which is a string-valued function that fetches the title to be found in the ith address in the file, and (b) SETTIT(i,s), where i refers to an address in the file and s is a string. Calling SETTIT(i,s) has the effect that afterward FETTIT will return the string s.

These assumptions are not very strong, but they allow one to write any application programs that do not need to make assumptions about the lengths of the strings, the possible contents of the strings, etc. An example would be a program that types out individualized form letters ("Dear Capt. Smith, We were pleased to learn that you have achieved the rank of Capt because now we can offer you...")*.

The assumptions enable us to define the abstract interface among all programs to be written to process the address lists by the informal specifications given in Fig. 6 or the formal specifications given in Fig. 7. Guttag [7] is a good introduction to reading the specifications in Fig. 7.

*For many programs one would make additional assumptions. For example, one might assume that if a military rank is given, then branch of service will be supplied, and that if a GS grade is provided, branch of service and a military rank will not be given. Such assumptions should be stated explicitly, because programs that make them might function incorrectly if the address file contains the address of a retired officer who still uses his title and branch of service but now holds a civil service job.

FUNCTION: ADDCITY(CTR,STR) **MODULE:** ASM

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new CITY field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in CITY field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDCORA(CTR,STR) **MODULE:** ASM [*ADD COMMAND OR ACTIVITY*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new CORA field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in CORA field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDGN(CTR,STR) **MODULE:** ASM [*ADD GIVEN NAME*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new GN field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in GN field of address CTR.
Error call if CTR < 1 or CTR FETNUM

Fig. 6 – Informal specifications

FUNCTION: ADDGSL(CTR,STR) **MODULE:** ASM *[ADD G S LEVEL]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new GSL field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in GSL field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDLN(CTR,STR) **MODULE:** ASM *[ADD LAST NAME]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new LN field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in LN field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDSERV(CTR,STR) **MODULE:** ASM *[ADD SERVICE BRANCH]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new SERV field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in SERV field of address CTR.
Error call if CTR < 1 or CTR FETNUM

Fig. 6 (Continued) – Informal specifications

FUNCTION: ADDSORP(CTR,STR) **MODULE:** ASM *[ADD STREET OR P.O. BOX]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new SORP field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in SORP field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDSTATE(CTR,STR) **MODULE:** ASM *[ADD STATE]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new STATE field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in STATE field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: ADDTIT(CTR,STR) **MODULE:** ASM *[ADD TITLE]*

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new TIT field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in TIT field of address CTR.
Error call if CTR < 1 or CTR FETNUM

Fig. 6 (Continued) – Informal specifications

FUNCTION: ADDZIP(CTR,STR) **MODULE:** ASM [*ADD ZIP* CODE]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed
STR	String	String to be stored as new ZIP field of address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Stores string STR in ZIP field of address CTR.
Error call if CTR < 1 or CTR FETNUM

FUNCTION: INIT **MODULE:** ASM [*INIT*IATE]

INPUT PARAMETERS: None

Name	Type	Description

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Initializes storage array by declaring size and dimension and setting flags for maximum number of addresses allowed

---

FUNCTION: FETBORC(CTR) **MODULE:** ASM [*FETCH BRANCH OR CODE*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: BORC field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

Fig. 6 (Continued) — Informal specifications

FUNCTION: FETCITY(CTR) **MODULE:** ASM [*FETCH CITY*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: CITY field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETCORA(CTR) **MODULE:** ASM [*FETCH COMMAND OR ACTIVITY*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: CORA field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETGN(CTR) **MODULE:** ASM [*FETCH GIVEN NAME*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: GN field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

Fig. 6 (Continued) – Informal specifications

FUNCTION: FETGSL(CTR) **MODULE:** ASM [*FETCH GS LEVEL*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: GSL field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETLN(CTR) **MODULE:** ASM [*FETCH LAST NAME*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: LN field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETSERV(CTR) **MODULE:** ASM [*FETCH SERVICE*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: SERV field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

Fig. 6 (Continued) – Informal specifications

FUNCTION: FETSORP(CTR) **MODULE:** ASM [*FET*CH *S*TREET *OR P*.O. BOX]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: SORP field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETSTATE(CTR) **MODULE:** ASM [*FET*CH *STATE*]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: String

FUNCTION VALUE: STATE field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: FETTIT(CTR) **MODULE:** ASM [*FET*CH *TIT*LE]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: TIT field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

Fig. 6 (Continued) — Informal specifications

FUNCTION: FETZIP(CTR) **MODULE:** ASM [*FET*CH *ZIP* CODE]

INPUT PARAMETERS:

Name	Type	Description
CTR	Integer	Number of address being accessed

FUNCTION VALUE TYPE: String

FUNCTION VALUE: ZIP field of address CTR

EFFECTS: Error call if CTR < 1 or CTR > FETNUM

FUNCTION: SETNUM(N) **MODULE:** ASM [*SET NUM*BER]

INPUT PARAMETERS:

Name	Type	Description
N	Integer	Number of addresses read

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Sets number of addresses actually stored.

FUNCTION: FETNUM **MODULE:** ASM [*FET*CH *NUM*BER]

INPUT PARAMETERS: None

Name	Type	Description

FUNCTION VALUE TYPE: Integer

FUNCTION VALUE: Number of addresses stored by ASM

EFFECTS: None

Fig. 6 (Continued) — Informal specifications

MODULE: ASM

SYNTAX

INIT:		→	asm *					
ADDTIT:	asm	X	integer	X	string	→	asm	*
ADDGN:	asm	X	integer	X	string	→	asm	*
ADDLN:	asm	X	integer	X	string	→	asm	*
ADDSERV:	asm	X	integer	X	string	→	asm	*
ADDBORC:	asm	X	integer	X	string	→	asm	*
ADDCORA:	asm	X	integer	X	string	→	asm	*
ADDSORP:	asm	X	integer	X	string	→	asm	*
ADDCITY:	asm	X	integer	X	string	→	asm	*
ADDSTATE:	asm	X	integer	X	string	→	asm	*
ADDZIP:	asm	X	integer	X	string	→	asm	*
ADDGSL:	asm	X	integer	X	string	→	asm	*
SETNUM:	asm	X	integer	→	asm *			
FETTIT:	asm	X	integer	→	string			
FETGN:	asm	X	integer	→	string			
FETGN:	asm	X	integer	→	string			
FETLN:	asm	X	integer	→	string			
FETSERV:	asm	X	integer	→	string			
FETBORC:	asm	X	integer	→	string			
FETCORA:	asm	X	integer	→	string			
FETSORP:	asm	X	integer	→	string			
FETCITY:	asm	X	integer	→	string			
FETSTATE:	asm	X	integer	→	string			
FETZIP:	asm	X	integer	→	string			
FETGSL:	asm	X	integer	→	string			
FETNUM:	asm	→	integer					

Fig. 7 — Formal specifications

Explicit parameters: Integer addr and string str

Implicit parameters:

Type	Name	Default
asm	asm	System asm

*Function value type: hidden from user.

Semantics:

V *integer* addr, V *asm* asm, V *string* str

FETα (INIT, addr) = error

If addr ≤ 0 *or* addr > FETNUM(asm)

then FETα(asm,addr) = error

else FETα (ADDβ(asm,addr,str), addr') =

if α ≠ β *or* addr ≠ addr'

then FETα (asm,addr')

else str

FETNUM(INIT) = error

FETNUM(SETNUM(asm,i)) = i

FETNUM(ADDα(asm,addr,str)) = FETNUM(asm)

FETNUM(FETα(asm,addr)) = FETNUM(asm)

where α and β range over the strings:

{'GN', 'LN', 'SERV', 'BORC', 'CORA', 'SORP', 'CITY', 'STATE', 'ZIP', 'TIT', 'GSL'}

Fig. 7 (Continued) – Formal specifications

These specifications have been used (in an NRL course on software engineering) to process address lists, of which some were in the format shown in Fig. 2 and others were in that shown in Fig. 3. The application programs were used without change for files of both formats. The file access functions defined in Figs. 6 and 7 were used without change by several application programs. One of the applications was a program to select the addresses within the Washington Military District (using the zip code); the other picked out VIPs, considering all officers of RANK 0-6 or higher and all "equivalent" civilians as VIPs. The method used was independent of the format of the input; use of the abstract interface allowed these programs to be procured without knowledge of the input format and to be used with two distinct file formats.

ON THE NEED FOR ASSUMPTIONS THAT ARE NOT SHARED BY ALL POSSIBLE INTERFACES

The assumptions that were listed and expressed in the specifications are sufficient for writing some of the application programs but may not be sufficient for all of them. For example, we have stated that the zip code will be a string and have not stated that the string is a nonnegative integer less than 10000. Any program that made this assumption (for example, by using the zip code as an index to a 10,000-element array) could not be demonstrated to be corect without stating this additional assumption. The price that one pays for making this assumption is that it rules out postal codes for many foreign countries. This may be an acceptable limitation; a program computing statistics on U.S. zip-code usage is inherently restricted in its applicability. We should however avoid making such an assumption unwittingly. The use of precise specifications and a programming-language compiler that performs type-checking [8,9] would detect many such errors early in the development of the system.

In procuring a set of programs using an abstract interface, where some of the programs need not make the additional assumptions, it is preferable that the additional information (the less abstract interface) not be supplied to those writing programs that can be written without it [3].

If two programs are based on more assumptions than the rest of the package, they need not necessarily make the same additional assumptions. One example would be a program that was designed to process British addresses to determine which county the addressee lived in. Such programs would want to assume that the postal code was a six-character string with the mnemonic coding used in Britain. Other programs might make the assumption that "title" was restricted to military ranks. Use of the phrase "less abstract" may be an abuse of language. It implies an ordering that may not exist.

COMPLETING THE SYSTEM BY STEPWISE ADDITION OF ASSUMPTIONS

As we stated earlier, the assumptions implicit in the abstract-interface specifications will not usually allow us to write a complete system*. To complete the system, we must make additional assumptions.

*The exception to this statement would be all of the possible interfaces being such that one can identify the actual interface by studying the input data.

Often one will complete the system in a single step, but sometimes it pays to proceed more slowly. Consider the address file specification and set of assumptions. They assume a previously defined data-type *string*. If the language or library does not include a string-manipulation facility, such a facility must be provided to complete the system. For any string implementation some assumptions must be made about the total number of strings in the system and their expected size. In applications like ours it is reasonable to assume a fixed upper bound for the length of a string. Using this additional information, one can define an interface to the string-manipulation package (Figs. 8 and 9). The implementation of these functions gives us a second component of our system. The first component will work for any format that meets our original assumptions. The second component assumes nothing about the order or content of the input information but assumes that no item need exceed a set length. Use of the combined package is predicated on both sets of assumptions, but should the maximum-length assumption prove wrong, only the string portion need be changed. The implementation of the FETCH and SET functions will then assume a complete description of the actual format including the fixed-length limitation.

One way that the additional assumptions may be made explicit in the definition of the abstract interface is by the introduction of functions that are not implementable unless the assumptions are met. For example, if we are willing to make the assumption that the address always determines the county in which the address is located, we may add the function COUNTY to the abstract interface. Programs that use COUNTY can be used only with input files such that the county can be determined from the data present.

WHERE IS THE SEMANTIC SPECIFICATION?

Notably absent from any of the assumptions or interface specifications in this report is any formal statement about the meaning of the information in the strings. Nothing in our interface forbids someone inserting a house number in the zip code field or interpreting a zip code as a house number.

Systems of the sort that we are discussing provide information storage, transmission, and retrieval functions. Any assumptions about the relation between the information in the system and phenomena in the outside world are agreements between those who insert data in the system and those who use data from the system. The system is not concerned with any aspects that it is not required to verify.

Any information about the meaning of the data (how it is obtained or what should be done with it) will not be part of the abstract (or internal) interface that we are discussing. The specifications for the larger system of which the embedded computer system is a part must include this additional information.

IMPLEMENTATION CONSIDERATIONS AND LANGUAGE LIMITATIONS

The specifications that are obtained by following the method proposed in this report assume the existence of data types that might not be built into the programming language being used. In the address example the specifications referred to a type (*string*) that is available in

NAME: BLANKSTRING **MODULE:** STM

SYNTAX

BLANKSTRING: → *string*

Explicit parameters: None

Implicit parameters: None

Function value type: *string*

SEMANTICS

Initial value: None

Function value: str *ϵ string*, such that

∀ i, 1 ⩽ i ⩽ MAXSTRING, RETCHAR(str,i) = ' '

Effects: None

NAME: INSERT **MODULE:** STM

SYNTAX

INSERT: *string* X *integer* X *char* → *string*

Explicit parameters: *string* str, *integer* loc, *char* c

Implicit parameters: None

Function value type: *string*

SEMANTICS

Initial value: None

Function value: str, such that

∀ i, 1 ⩽ i ⩽ MAXSTRING, *if* i = loc, *then* CHAREQ(c,RETCHAR(str,i)) = TRUE

else CHAREQ(RETCHAR(str,i),RETCHAR(str',i)) = TRUE

Effects: *If* loc < 1 *or* loc > MAXSTRING, *then* ERRORCALL

Fig. 8 — Formal specification of the string modules (STM)

NAME: MAXSTRING **MODULE:** STM

SYNTAX

MAXSTRING → *integer*

Explicit parameters: None

Implicit parameters: None

Function value type: *integer*

SEMANTICS

Initial value: None

Function value: k, such that $\forall i$, $1 \leqslant i \leqslant k$,
CHAREQ(RETCHAR(BLANKSTRING,i), ' ') = TRUE

and $\forall j$, $j > k$, $\forall s \in$ *string*
RETCHAR(s,j) is undefined

Effects: None

NAME: RETCHAR **MODULE:** STM

SYNTAX

RETCHAR: *string* X *integer* → *char*

Explicit parameters: *string* str, *integer* loc

Implicit parameters: None

Function value type: *integer*

SEMANTICS

Initial value: None

Function value: $\forall$ i, j $1 \leqslant$ i, j $\leqslant$ MAXSTRING,
RETCHAR(BLANKSTRING,i) = ' '
RETCHAR(INSERT(str,i,c),j) = *if* i = j

then c

else RETCHAR(str,j)

Fig. 8 (Continued) – Formal specification of the string module (STM)

FUNCTION CALLING FORM: BLANKSTRING(NEW) **MODULE:** STM

INPUT PARAMETERS:

Name	Type	Description
NEW	*String*	Fixed-length string consisting of blanks which is returned

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Initializes a string, NEW, to blanks and returns it

FUNCTION CALLING FORM: INSERT(STR,LOC,CHR) **MODULE:** STM

INPUT PARAMETERS:

Name	Type	Description
STR	*String*	String to be processed
LOC	*Integer*	Position in string to insert character
CHR	*Char*	Character to be inserted

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: Replaces LOC*th* character of STR with CHR. Error call if LOC $<$ 1 or LOC $>$ MAXSTRING

FUNCTION CALLING FORM: MAXSTRING **MODULE:** STM

INPUT PARAMETERS: None

Name	Type	Description

FUNCTION VALUE TYPE: *Integer*

FUNCTION VALUE: Length of strings

EFFECTS: None

FUNCTION CALLING FORM: RETCHAR(STR,LOC) **MODULE:** STM

INPUT PARAMETERS:

Name	Type	Description
STR	*String*	String to be accessed
LOC	*Integer*	Location in string of character sought

FUNCTION VALUE TYPE: *Char*

FUNCTION VALUE: The character in the LOC*th* position in string STR.

EFFECTS: Error call if LOC $<$ 1 or LOC $>$ MAXSTRING

Fig. 9 — Informal specification of the string module.

some languages but unavailable or available with strong limitations in other languages. In other examples we would write specifications in terms of even more specialized types (such as dates) that would certainly not be built into a language.

The assumption of the existence of a data type is the assumption of the existence of data elements and operators. If they are not built into the language, there is no reason the operators cannot be provided by means of macros or subroutines. However our specifications go further — they assume the ability to pass objects or values of the special type to procedures and the ability to define procedures that return values of this type.

If the language that must be used does not provide data-type extensibility, there is always a subterfuge by which the ability to pass and return values of the new type can be simulated. The method chosen will depend on the language available, the relative importance of time and space efficiency, and security considerations. Whatever the method chosen, it will require the adoption of certain programming conventions. (Often these conventions cannot be enforced by the compiler.) The adoption of these conventions will sometimes be simply a further refinement of the abstract interface (making futher assumptions); in other cases it will require a *change* in the syntax for calls on the functions defined by the specifications. The semantics of the functions specified will stay the same, but parameters may be passed through common blocks, global variables, pointers, etc.

The need to refine or alter our interface to accommodate the limitations of our present programming tools does not contradict the validity of the proposed method. In the actual implementation the syntax used for calling functions and communicating information between calling program and called program may be different from that specified during the design of the interface, but the information to be passed will not be affected.

CONCLUDING REMARKS

The purpose of this report has been to explain and demonstrate a systematic software-procurement procedure by means of which one may isolate the bulk of the software from changes in the actual external interface. If the specifications of the abstract interface are used to define the responsibility of the organization that delivers the internal software, that organization is effectively prevented from producing a system that is tied to any one particular interface. Moreover they must deliver a program that can be "completed" by persons without knowledge of internal details.

There are additional benefits. This process leads toward what is sometimes referred to as a cleaner structure of the software — one in which there is a good separation of concerns, allowing each component to be simpler and more easily understood. Further, those components that are not cognizant of the real-world details of the interface can be more elegant and subject to a more mathematical analysis. Elegance is not a property of systems that must deal with ugly real world facts, but it can be obtained in those components that are separated from the real world by the use of well-defined interfaces.

REFERENCES

1. D.L. Parnas, "On the Design and Development of Program Families," IEEE Transactions on Software Engineering SE-2 (No. 1), 1-9 (Mar. 1976).

2. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM **15** (No. 12), 1053-1058 (Dec. 1972).
3. D.L. Parnas, "Information Distribution Aspects of Design Methodology, Proceedings," 1971 IFIP Congress, North Holland Publishing Co.
4. D.L. Parnas, "A Technique for Software Module Specification with Examples," Communications of the ACM **15** (No. 5), 330-336 (May 1972).
5. E.W. Dijkstra, C.A.R. Hoare, and O.J. Dahl, *Structured Programming*, Academic Press, London, 1972.
6. N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM **14** (No. 4), 221-227 (Apr. 1971).
7. J. Guttag, "Abstract Data Types and the Development of Data Structures," SIGPLAN, SIGMOD Conference on DATA: Abstraction Definition and Structure (to be published in Communications of the ACM).
8. D.L. Parnas, J. Shore, and D. Weiss, "Abstract Types Defined as Classes of Variables," Proceedings of ACM Conference on DATA: Abstraction, Definition, and Structure, March, 1976.
9. CS-4 Language Reference Manual and Operating System Interface, Oct. 1975, Intermetrics, Inc.

ACKNOWLEDGMENTS

Discussions with H. Elovitz, John Guttag, John Shore, Barbara Trombka, and David Weiss have contributed a great deal to this report. Questions raised by H. Elovitz in an effort to apply these concepts led to especially significant changes.